CS61A Lecture 12
Immutable Trees

Jom Magrotker
UC Berkeley EECS

July 9, 2012

Computer Science in the News

Soap bubble screen is 'the world's thinnest display'

The team says the display is the world's thinnest transparent screen

http://www.bbc.com/news/technology-18671061

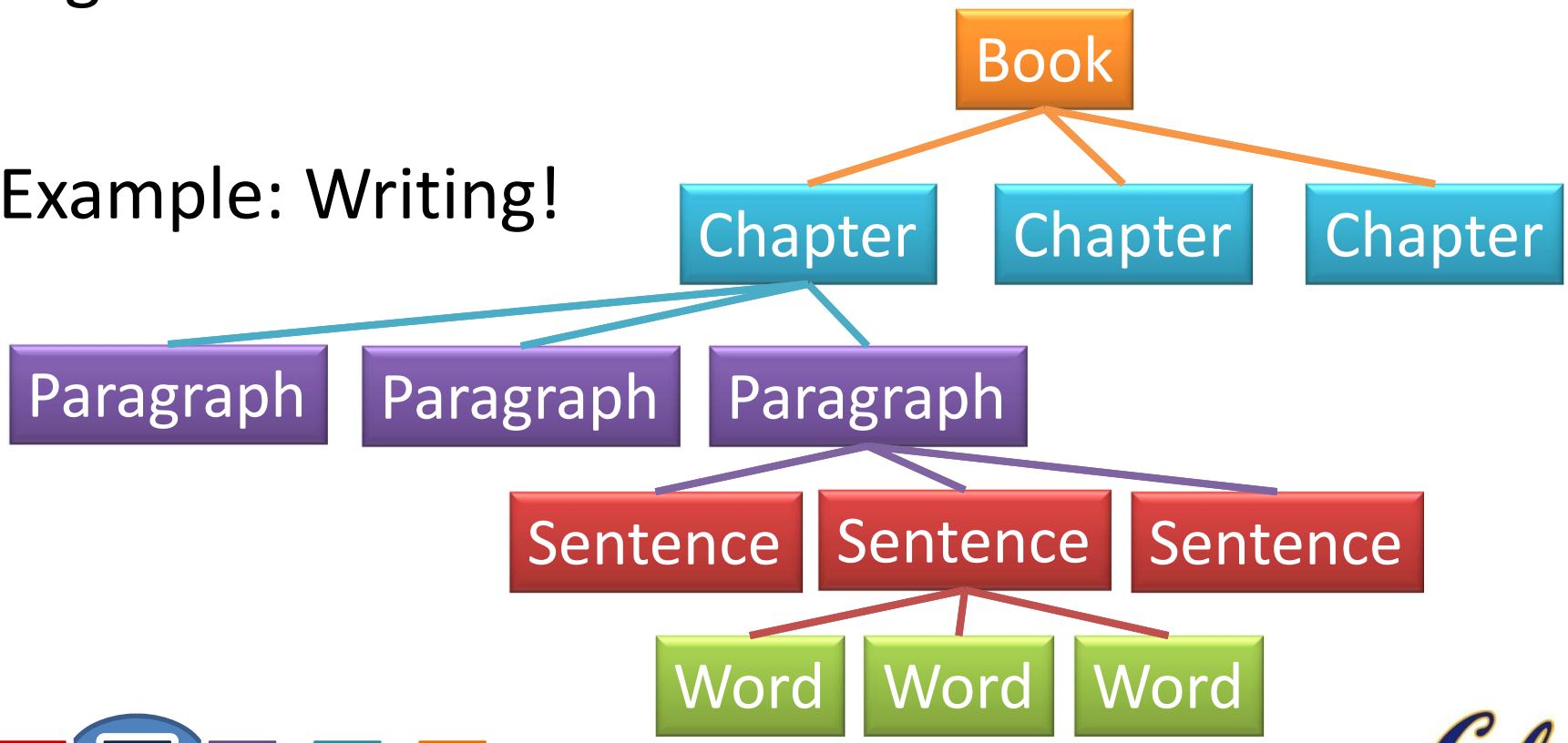
TODAY

- Review: Immutable Trees
- Binary Trees and Binary Search Trees
- Extra: Tuples versus IRLists

Review: Hierarchical Data

Often, we find that information is nicely organized into hierarchies.

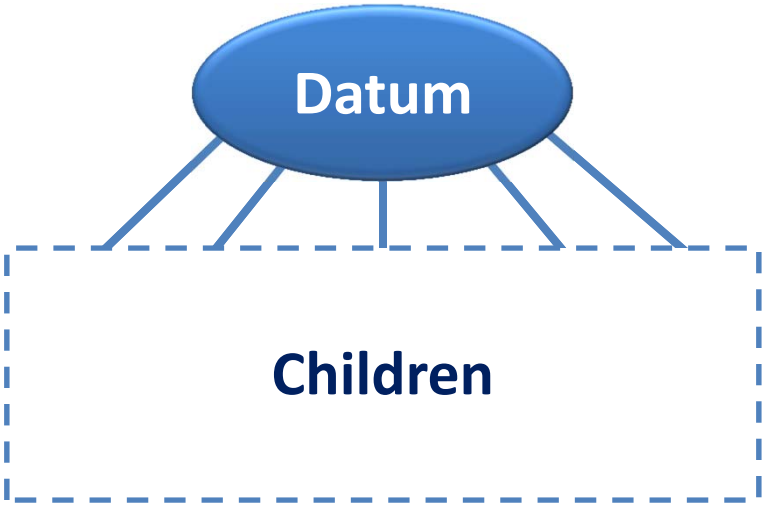
Example: Writing!



Review: Trees

A ***tree*** data structure traditionally has two parts:

1. A ***datum***: Information stored at the top.
2. Some ***children***: Trees that appear below this tree.



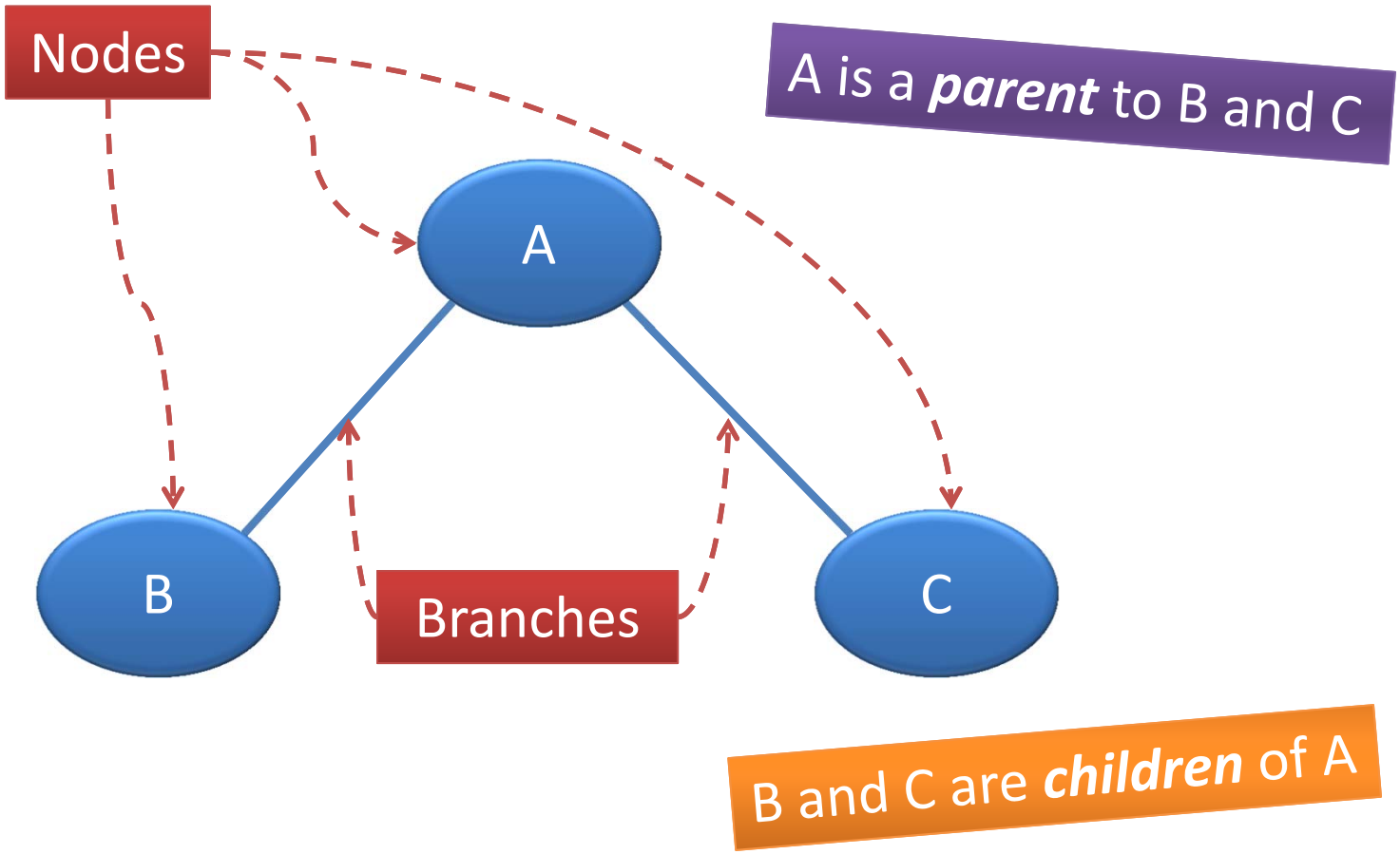
REVIEW: TREES

Notice that trees are also *recursively defined*.

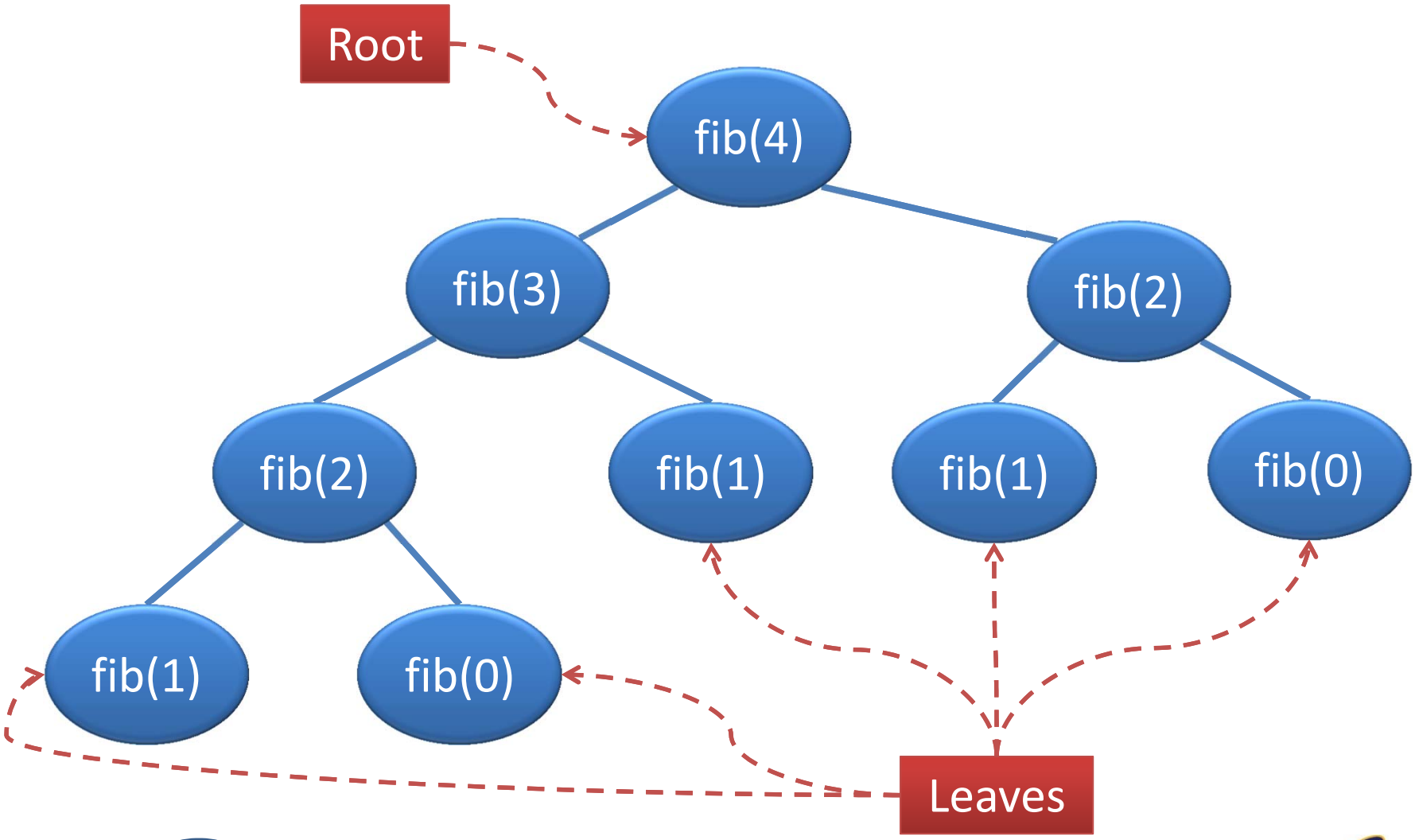
A tree is made from other trees –

these trees are its *subtrees*.

Review: Trees



Review: Trees



Review: Immutable Trees

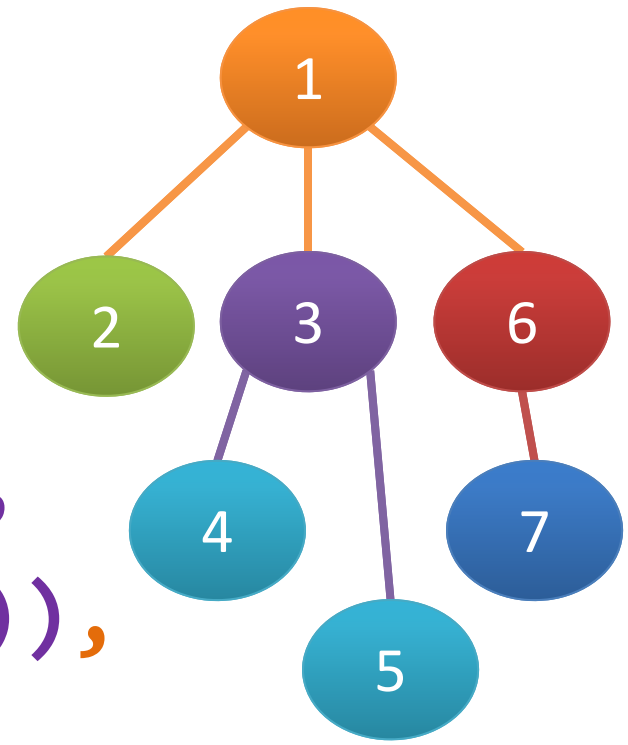
```
make_itree(1,
    (make_itree(2),
     make_itree(3,
        (make_itree(4),
         make_itree(5))),
     make_itree(6,
        (make_itree(7),))))
```

Root

No children

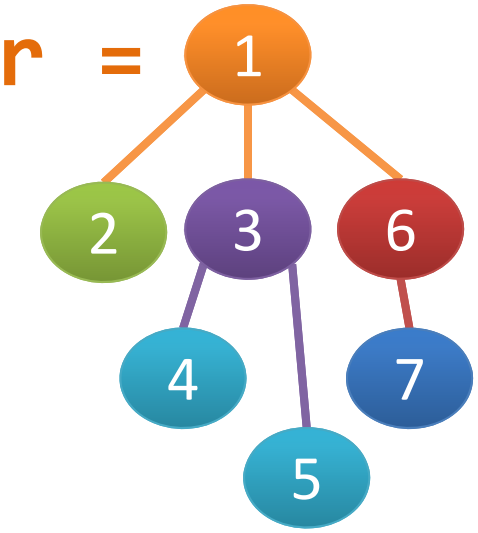
Tuple of children
(each is an ITree!)

Only one child



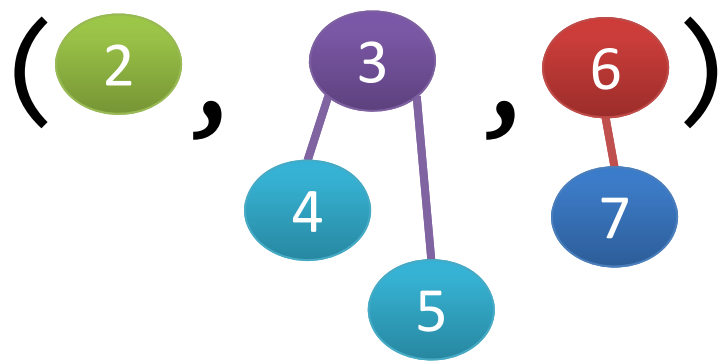
Review: Immutable Trees

`fir` =



`itree_datum(fir)` is 1

`itree_children(fir)` is the tuple



This is not what is printed! The expression *evaluates* to a tuple of ITrees, which are the children of `fir`.

A group of trees is called a ***forest***.

Review: ITrees

```
def make_itree(datum, children=()):
    return (datum, children)

def itree_datum(t):
    return t[0]

def itree_children(t):
    return t[1]
```

Constructor

Selectors

Review: Operating on ITrees

Write the function `itree_prod`, which takes an ITree of numbers and returns the product of all the numbers in the ITree.

```
>>> t = make_itree(1, (make_itree(2),
                       make_itree(3),
                       make_itree(4)))
>>> itree_prod(t)
24
```

Example: Operating on ITrees

Idea: Split the problem into two different parts: one that handles a single tree and one that handles a forest.

```
def itree_prod(t):
    return itree_datum(t) * forest_prod(itree_children(t))

def forest_prod(f):
    if len(f) == 0:
        return 1
    return itree_prod(f[0]) * forest_prod(f[1:])
```

Returns the product of numbers in an ITree.

Datum, multiplied by…

… the product of all the numbers in all the ITrees in the forest.

These ITrees are the children.

Product of numbers in the first ITree, multiplied by…

… the product of all the numbers in all the other ITrees of the forest.

Example: Operating on ITrees

Idea: Split the problem into two different parts: one that handles a single tree and one that handles a forest.

```
def itree_prod(t):
    return itree_datum(t) * forest_prod(itree_children(t))
```

```
def forest_prod(f):
    if len(f) == 0:
        return 1
    return itree_prod(f[0]) * forest_prod(f[1:])
```

Not a data abstraction violation.
We *know* that f is a tuple of ITrees.

Example: Operating on ITrees

Idea: Split the problem into two different parts: one that handles a single tree and one that handles a forest.

```
def itree_prod(t):
    return itree_datum(t) * forest_prod(itree_children(t))
```

This is called ***mutual recursion*** because it involves two functions recursively calling each other!

```
def forest_prod(f):
    if len(f) == 0:
        return 1
    return itree_prod(f[0]) * forest_prod(f[1:])
```

Mutual Recursion

To find the product of the whole ITree in **`itree_prod`**, we need to apply **`itree_prod`** itself to the smaller subtrees that are the children of the provided itree.

`forest_prod` does this for us.

Example: Operating on ITrees

An alternate definition for `itree_prod` does not need two separate and smaller functions:

```
from operator import mul
def itree_prod(t):
    return itree_datum(t) * \
           reduce(mul,
                  map(itree_prod,
                      itree_children(t)),
                  1)
```

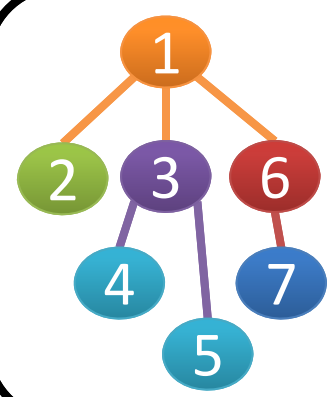
Applies `itree_prod` recursively on the children.

Multiplies all the results of applying `itree_prod` on the children.

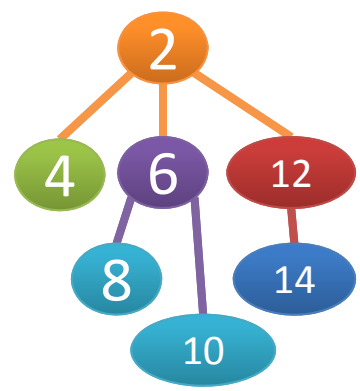
Practice: Operating on ITrees

Write a function `scale_itree` that takes an ITree of numbers and a scaling factor, and returns a new ITree that results from scaling each number in the original ITree.

`scale_itree(`



`, 2) =`



PRACTICE: OPERATING ON ITREES

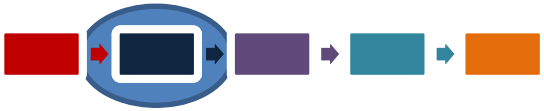
```
def scale_itree(itree, factor):
    return make_itree(____________________________,
                      scale_forest(______________________,
                                   ______))

def scale_forest(forest, factor):
    results = ()
    for itree in forest:
        results = results + ___________________________
    return results
```

PRACTICE: OPERATING ON ITREES

```
def scale_itree(itree, factor):
    return make_itree(itree_datum(itree) * factor,
                      scale_forest(itree_children(itree),
                                   factor))

def scale_forest(forest, factor):
    results = ()
    for itree in forest:
        results = results + scale_itree(itree, factor)
    return results
```



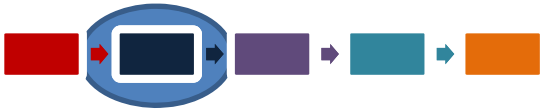
Practice: Operating on ITrees

Alternative solution:

```
def scale_itree(itree, factor):
    return make_itree(itree_datum(itree) * factor,
                      scale_forest(itree_children(itree),
                                   factor))

def scale_forest(forest, factor):
    if len(forest) == 0:
        return ()
    return (scale_itree(forest[0], factor),) + \
           scale_forest(forest[1:], factor)
```

Tuple of one tree



PRACTICE: OPERATING ON ITREES

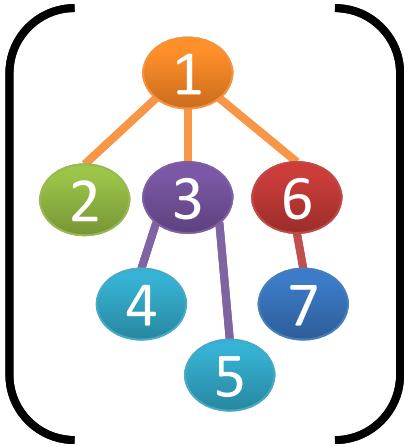
```
def scale_itree(itree, factor):
    return make_itree(____________________________,
                      tuple(map(__________,
                                ______________________)))
```

PRACTICE: OPERATING ON ITREES

```
def scale_itree(itree, factor):
    return make_itree(itree_datum(itree) * factor,
                      tuple(map(scale_itree,
                                itree_children(itree))))
```

Practice: Operating on ITrees

Write a function `count_leaves` that counts the number of leaves in the ITree provided.

`count_leaves` 

= 4

PRACTICE: OPERATING ON ITREES

```
def count_leaves(itree):
    if ______________:
        return 1
    return cl_forest(______________________)

def cl_forest(f):
    if len(f) == 0:
        return _
    return ____________________________________
```

Practice: Operating on ITrees

```
def count_leaves(itree):
    if is_leaf(itree):
        return 1
    return cl_forest(itree_children(itree))

def cl_forest(f):
    if len(f) == 0:
        return 0
    return count_leaves(f[0]) + cl_forest(f[1:])
```

```
def is_leaf(itree):
    return len(itree_children(itree)) == 0
```

PRACTICE: OPERATING ON ITREES

```
def count_leaves(itree):
    if ______________:
        return 1
    return reduce(___,
                  map(____________,
                      ________________________),
                  _)
```

Practice: Operating on ITrees

```
def count_leaves(itree):
    if is_leaf(itree):
        return 1
    return reduce(add,
                  map(count_leaves,
                      itree_children(itree)),
                  0)
```

ANNOUNCEMENTS

- Homework 6 is due tomorrow, **July 10**.
  - Starting with the next homework, we will mark questions as *core* and *reinforcement*.
  - The *core* questions are ones that we suggest you work on to understand the idea better.
  - The *reinforcement* questions are extra problems that you can practice with, but are not as critical.
- Project 2 is due on Friday, **July 13**.
- Homework 0 for the staff is now available.

ANNOUNCEMENTS

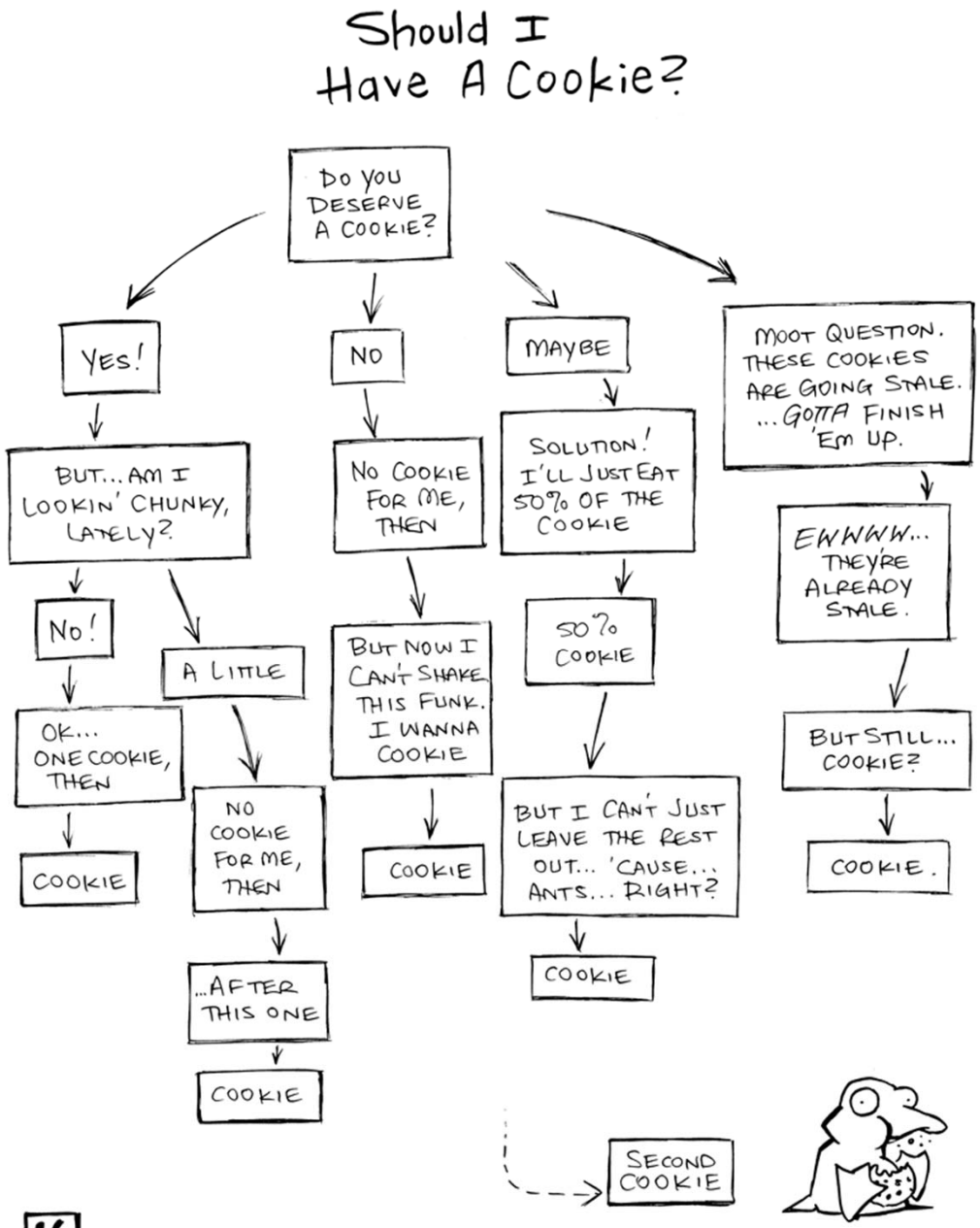
- Midterm 1 is **tonight**.
  - *Where*? 2050 VLSB.
  - *When*? 7PM to 9PM.
- Closed book and closed electronic devices.
- One 8.5” x 11” ‘cheat sheet’ allowed.
- Group portion is 15 minutes long.
- Post-midterm potluck on Wednesday, **July 11** in the **Wozniak Lounge** from **6:30pm to 10pm**.
  - Bring food, drinks and board games!
  - Drop by if and when you can.

ANNOUNCEMENTS

http://berkeley.edu/map/maps/ABCD345.html

CAMPANILE

BREAK

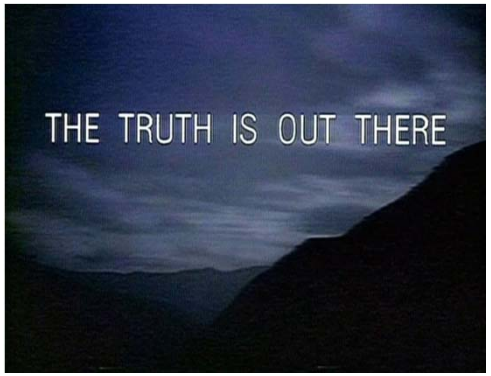


http://www.sheldoncomics.com/archive/101026.html/

Problem Solving: Searching

Searching is a problem that shows up frequently in many different (and daily!) settings.

Problem: We have a collection of data and we need to find a certain datum.



Examples:

- Searching for a person in a phone book.
- Searching for a webpage on the Internet.

… and so on.

http://www.spacepub.com/wp-content/uploads/2011/06/The-Truth-Is-Out-There.jpg

Problem Solving: Searching

Write a predicate function `has_num` that determines if a number is in a tuple of numbers.

```
def has_num(tup, num_to_find):
    for item in tup:
        if ___________:
            return ____
    return _____
```

Problem Solving: Searching

Write a predicate function has_num that determines if a number is in a tuple of numbers.

```
def has_num(tup, num_to_find):
    for item in tup:
        if item == num:
            return True
    return False
```

Problem Solving: Searching

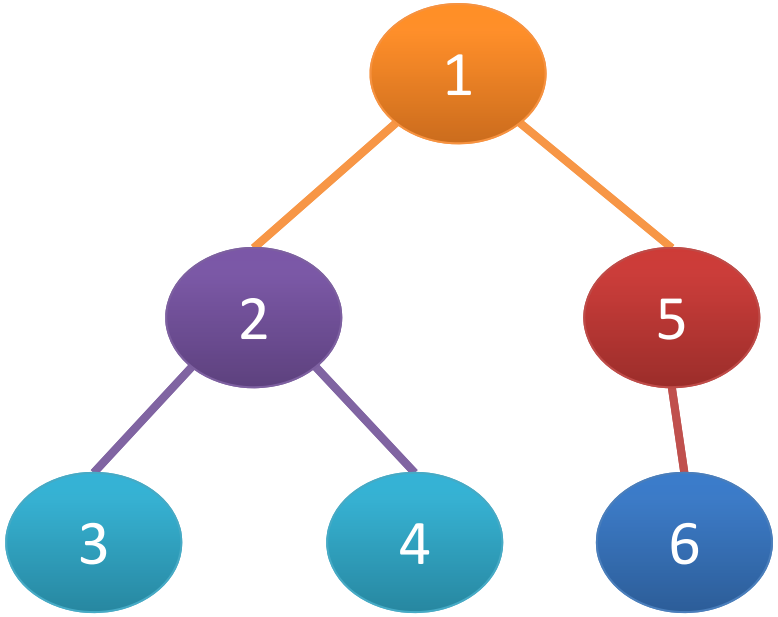
How does the running time of `has_item` scale as n, the length of the tuple, scales?

Can we do better?

http://i2.kym-cdn.com/photos/images/original/000/221/792/cute-puppy-pictures-anything-you-can-do-i-can-do-better.jpg

Trees in Real Life: Binary Trees

Trees where each node has *at most* two children are known as ***binary trees***.

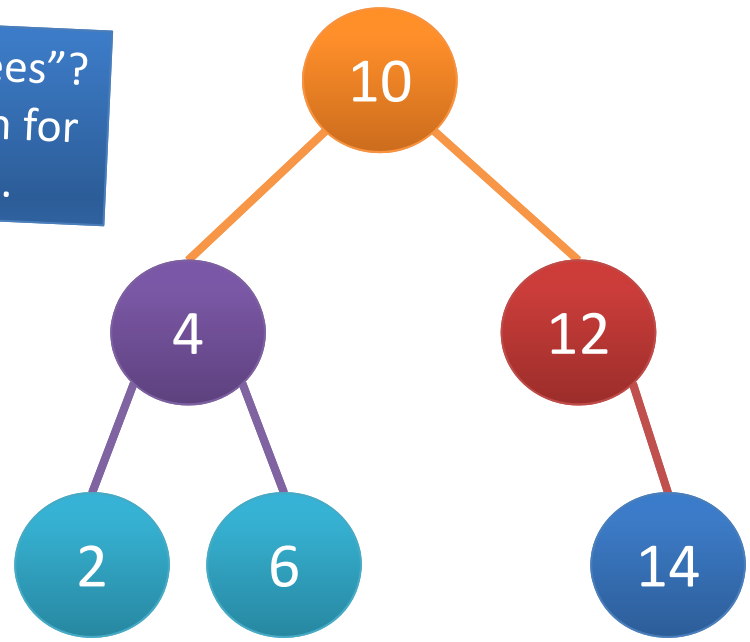


Trees in Real Life: Binary Search Trees

Binary search trees are binary trees where all of the items to the ***left*** of a node are ***smaller***, and the items to the ***right*** are ***larger***.

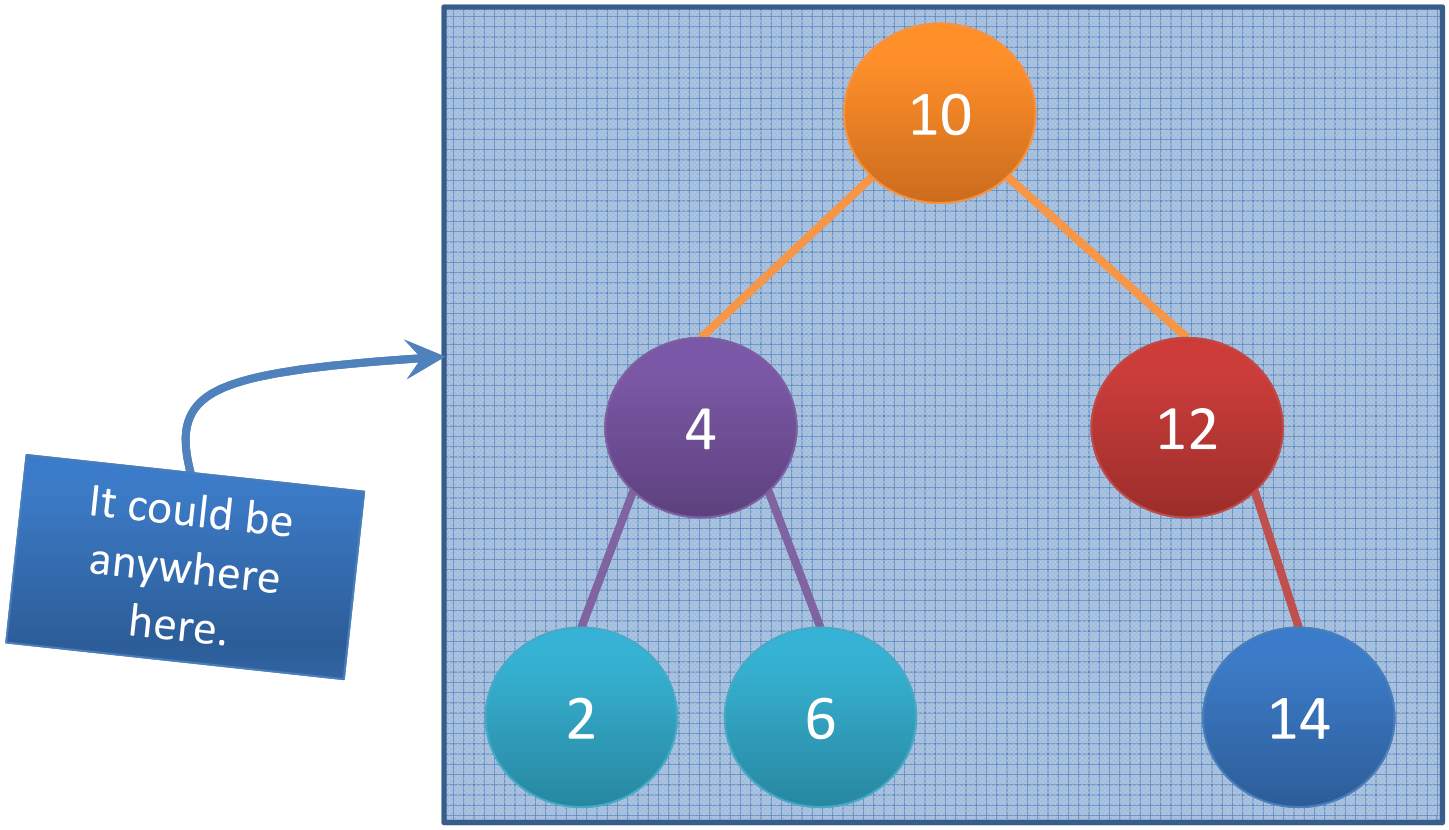
Why the term "*search* trees"? They speed up the search for an item in a sequence.

How?



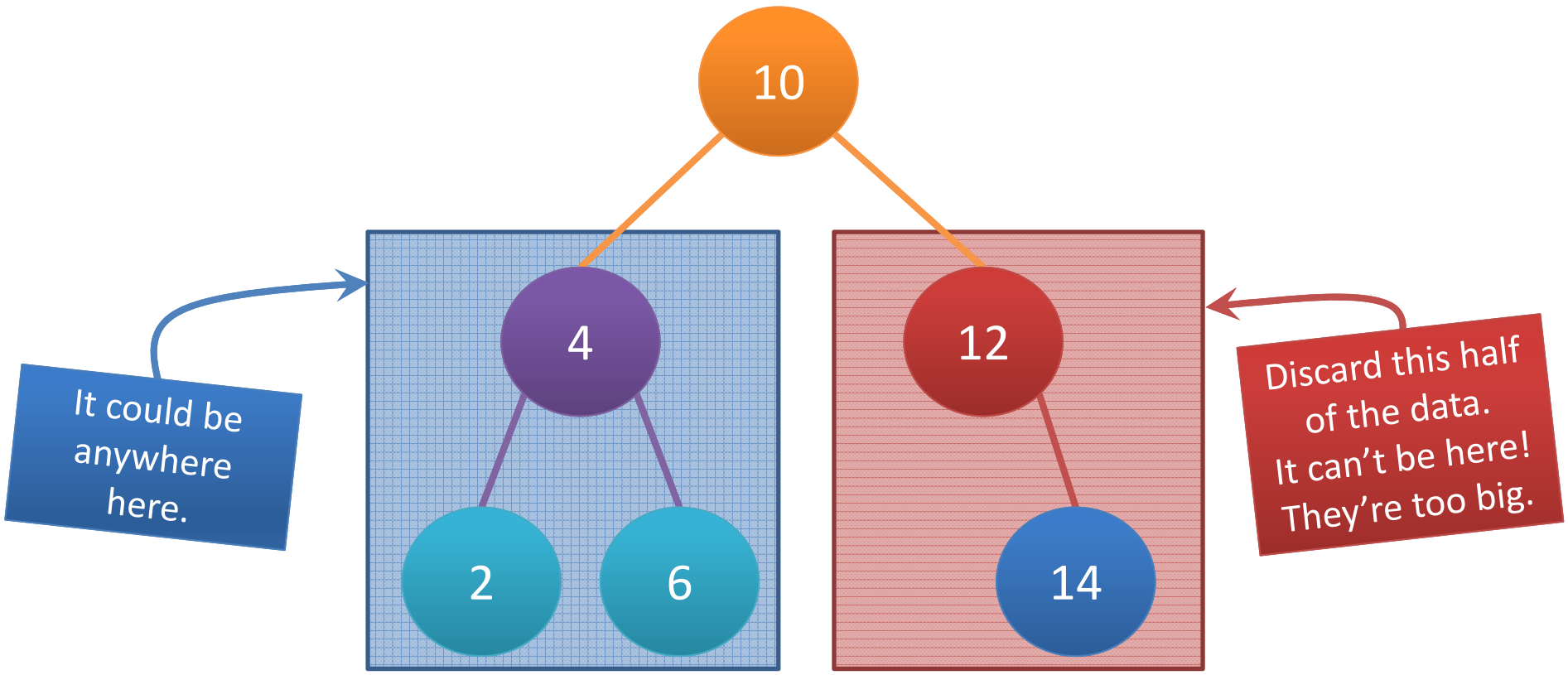
Trees in Real Life: Binary Search Trees

Let us try to find 4 in this binary search tree:



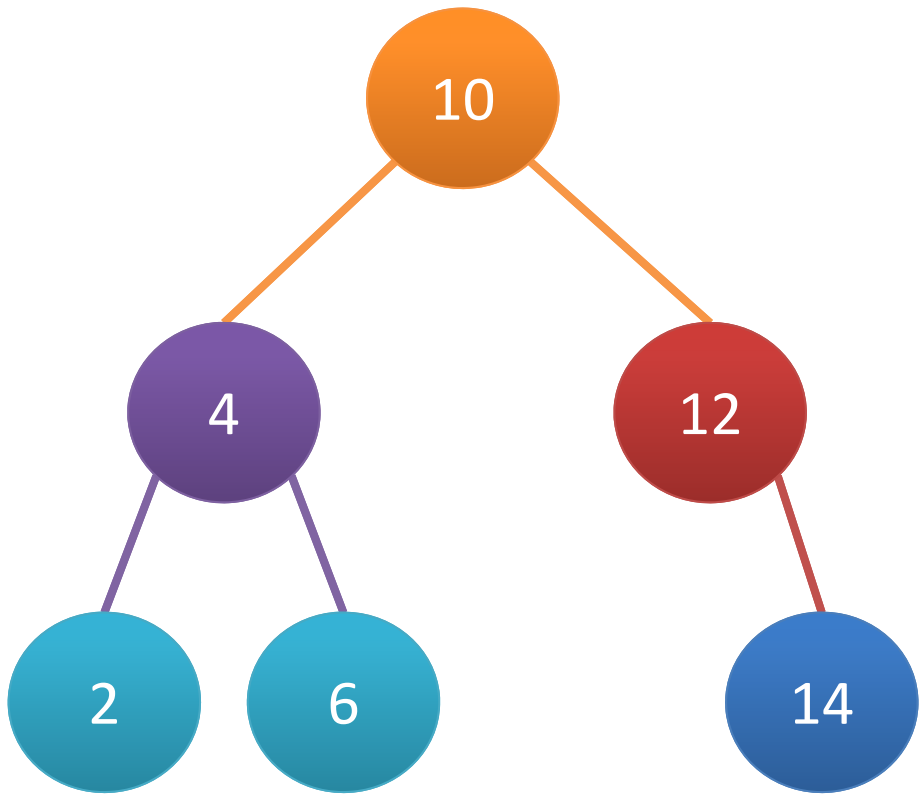
Trees in Real Life: Binary Search Trees

Let us try to find 4 in this binary search tree:



Trees in Real Life: Binary Search Trees

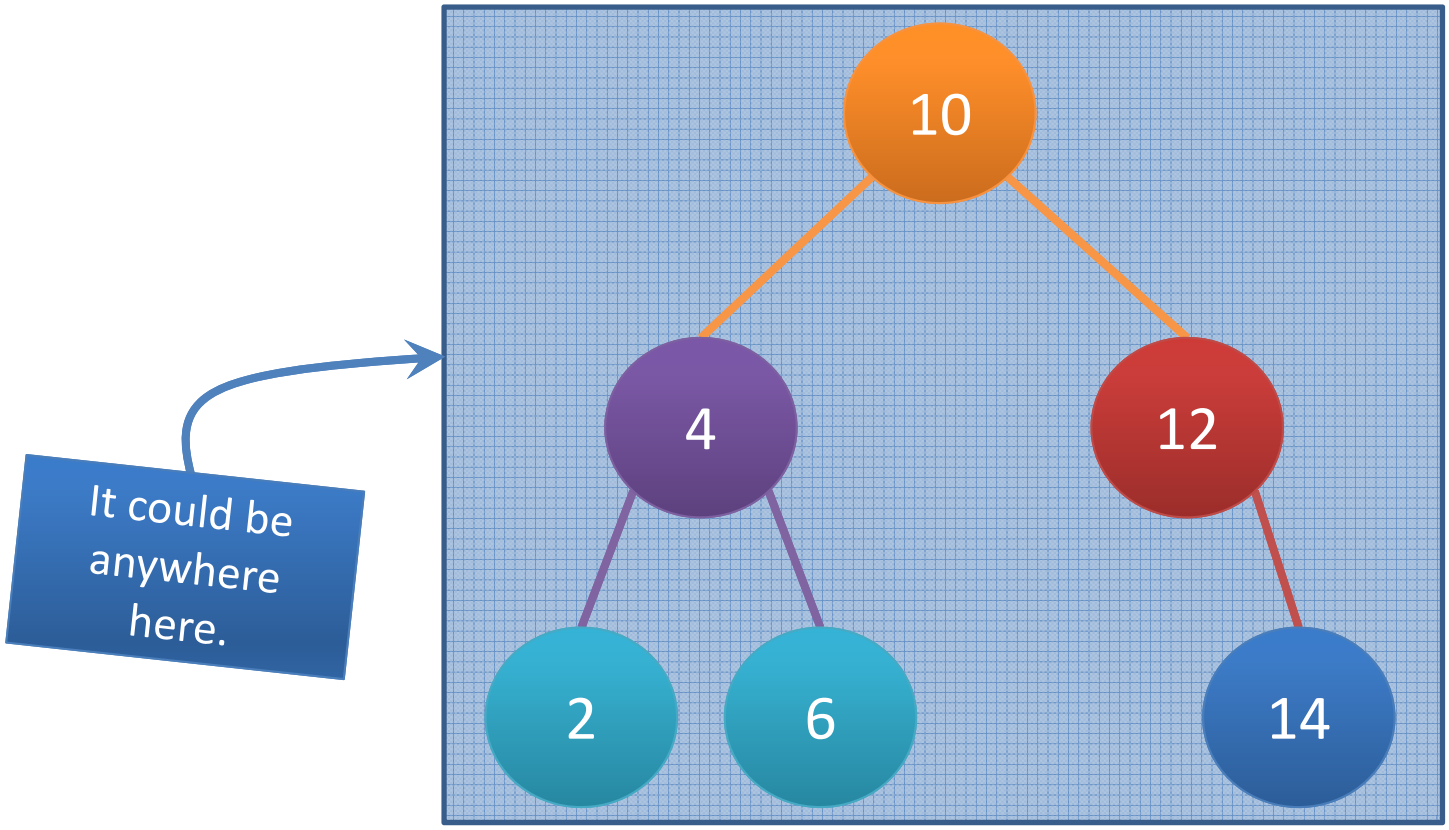
Let us try to find 4 in this binary search tree:



It is present in the BST. ☺

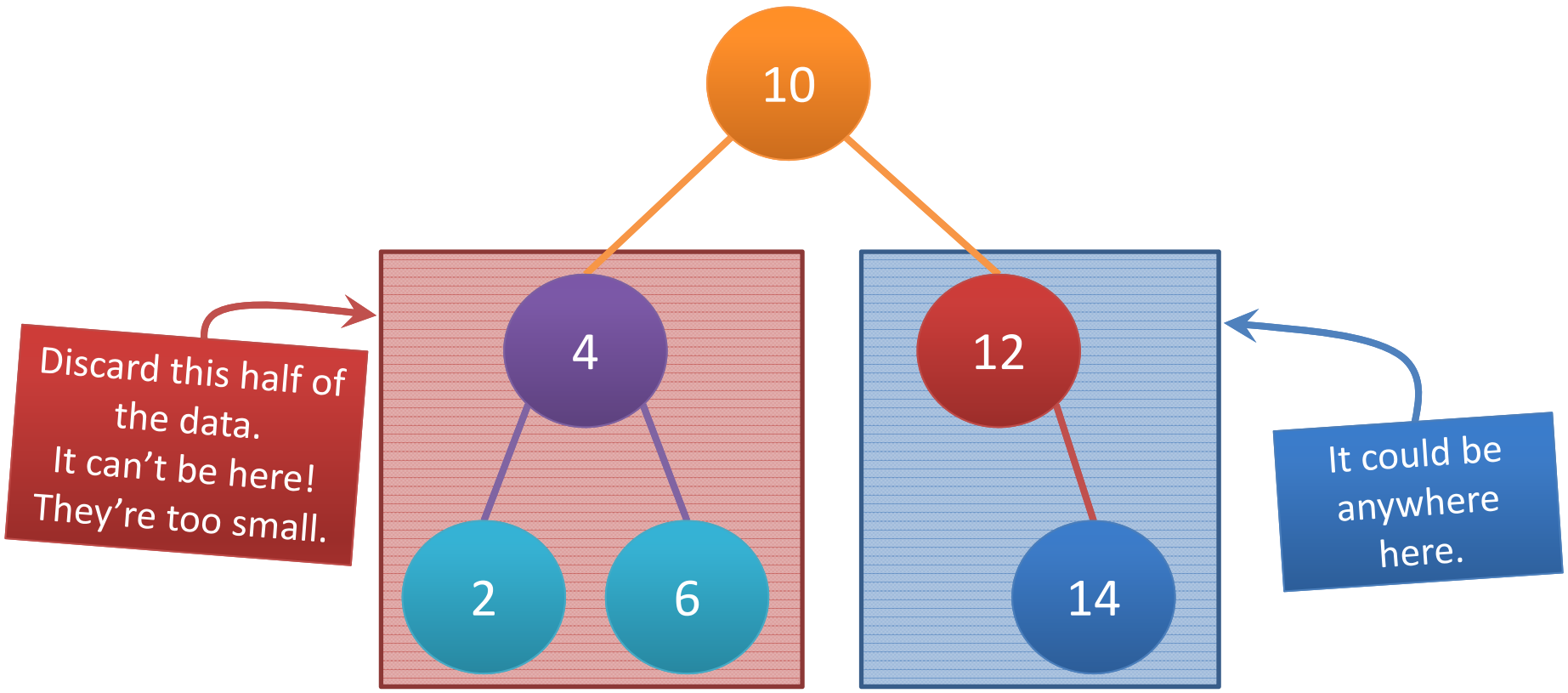
Trees in Real Life: Binary Search Trees

Let us try to find 14 in this binary search tree:



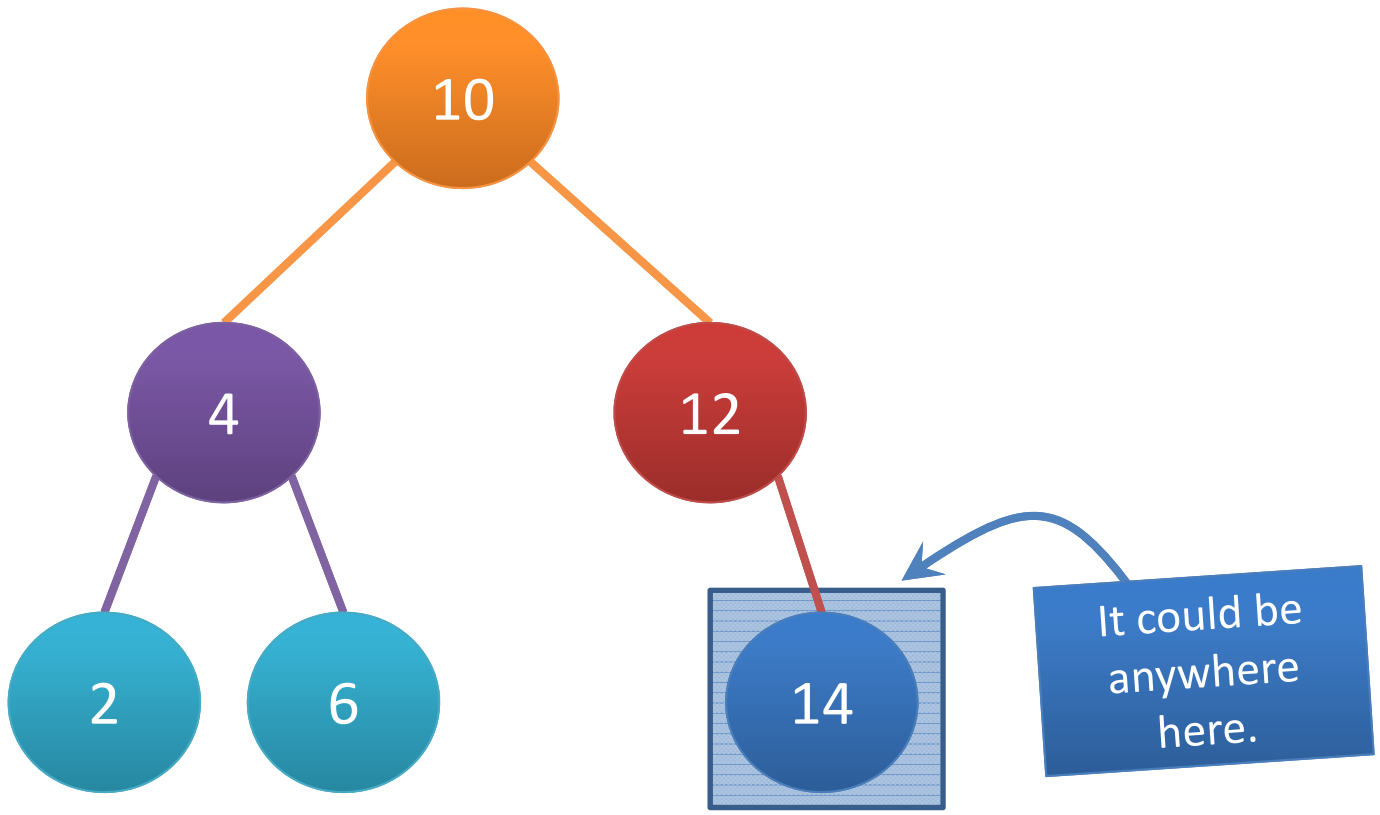
Trees in Real Life: Binary Search Trees

Let us try to find 14 in this binary search tree:



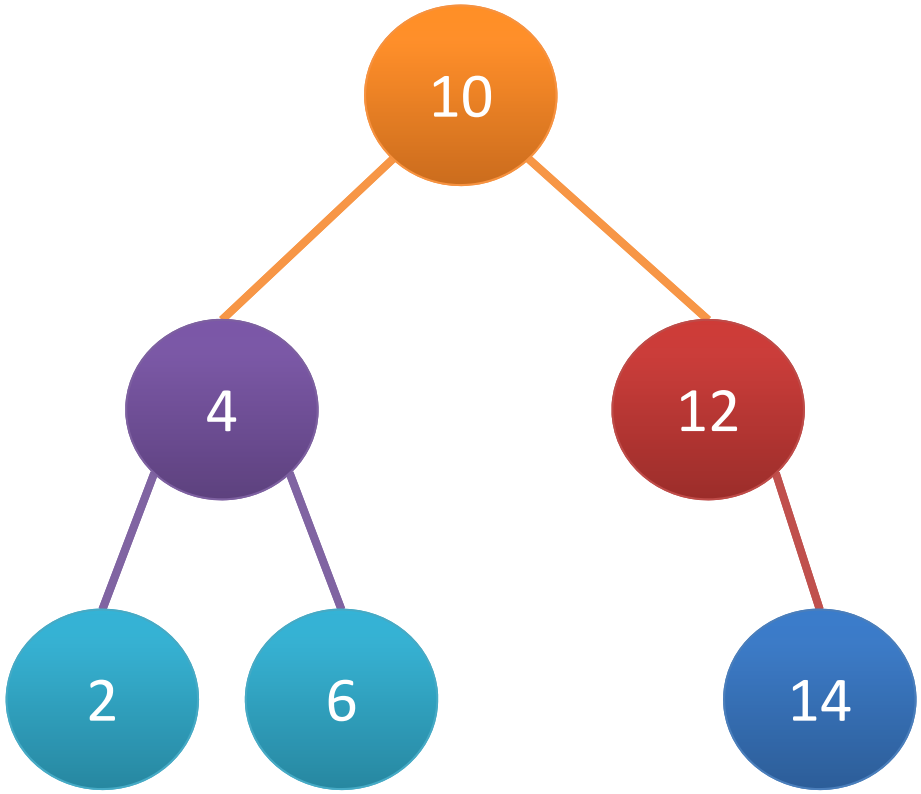
Trees in Real Life: Binary Search Trees

Let us try to find 14 in this binary search tree:



Trees in Real Life: Binary Search Trees

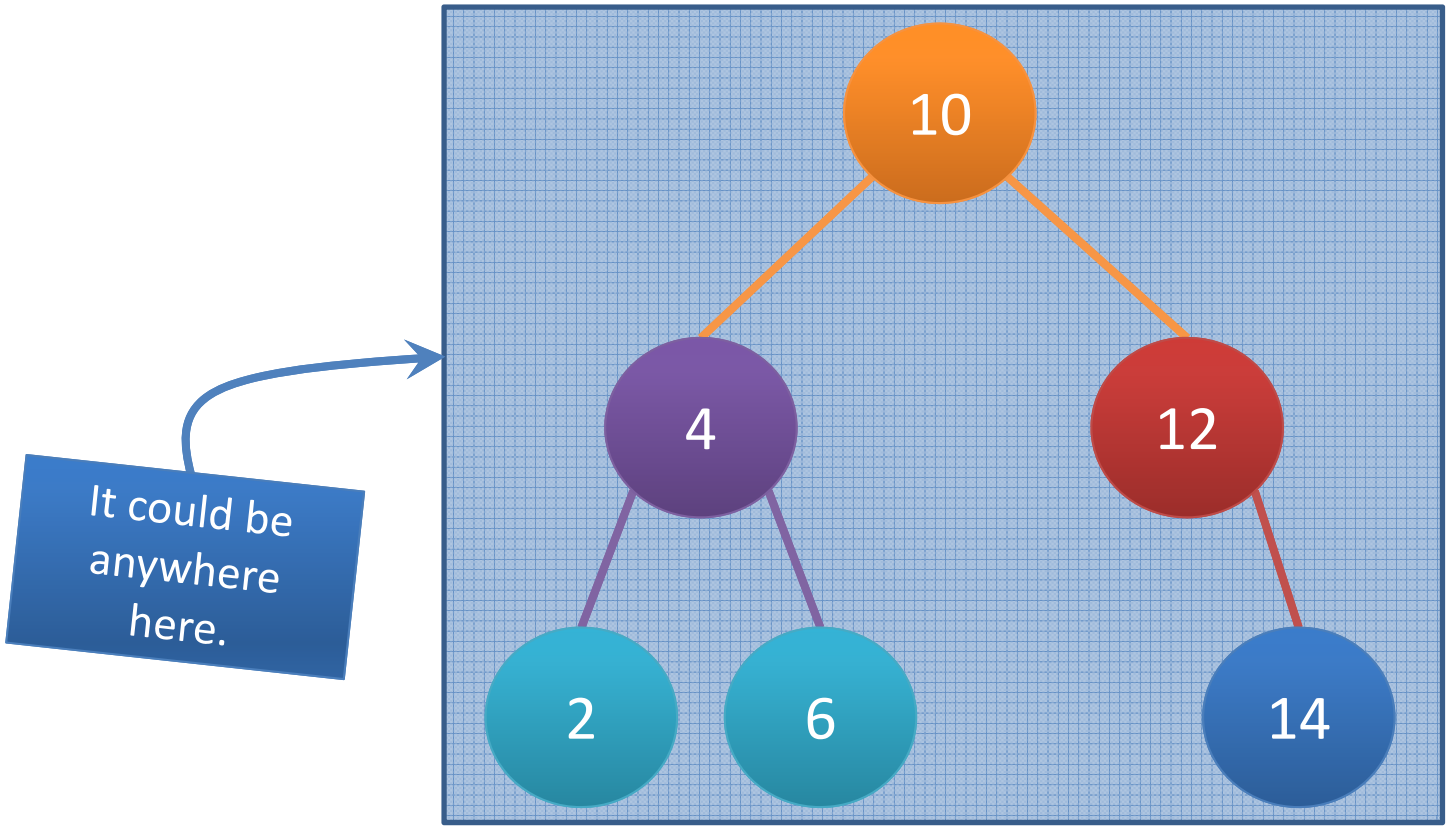
Let us try to find 14 in this binary search tree:



It is present in the BST. ☺

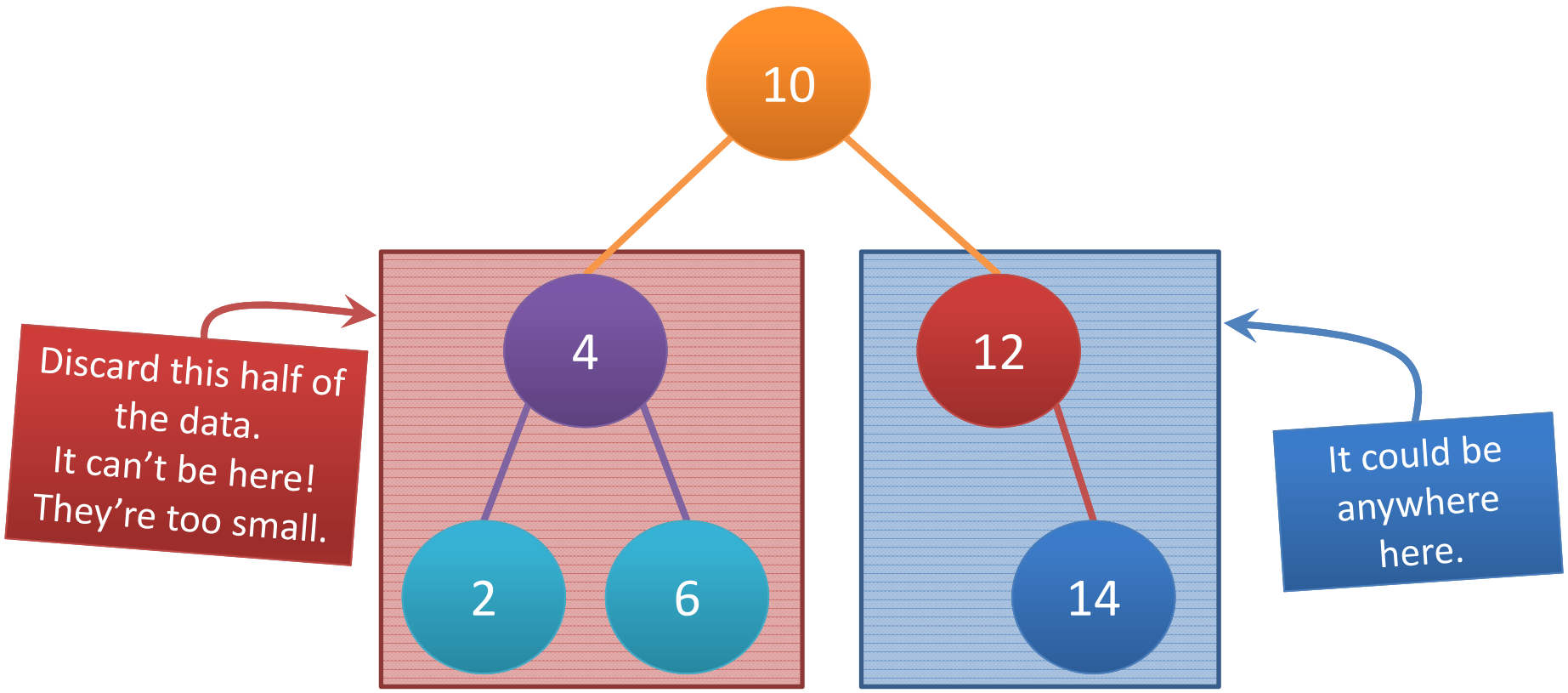
Trees in Real Life: Binary Search Trees

Let us try to find 11 in this binary search tree:



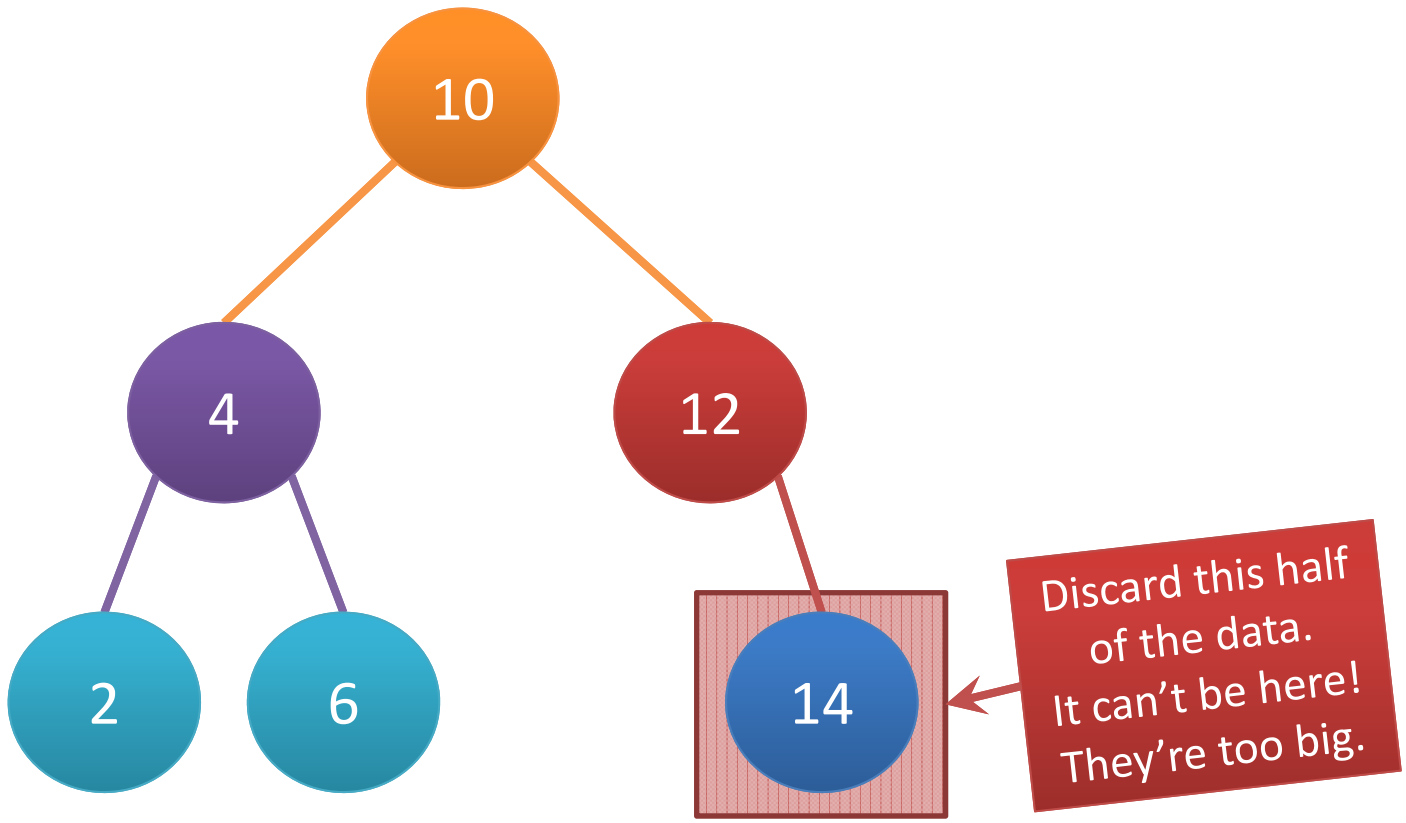
Trees in Real Life: Binary Search Trees

Let us try to find 11 in this binary search tree:



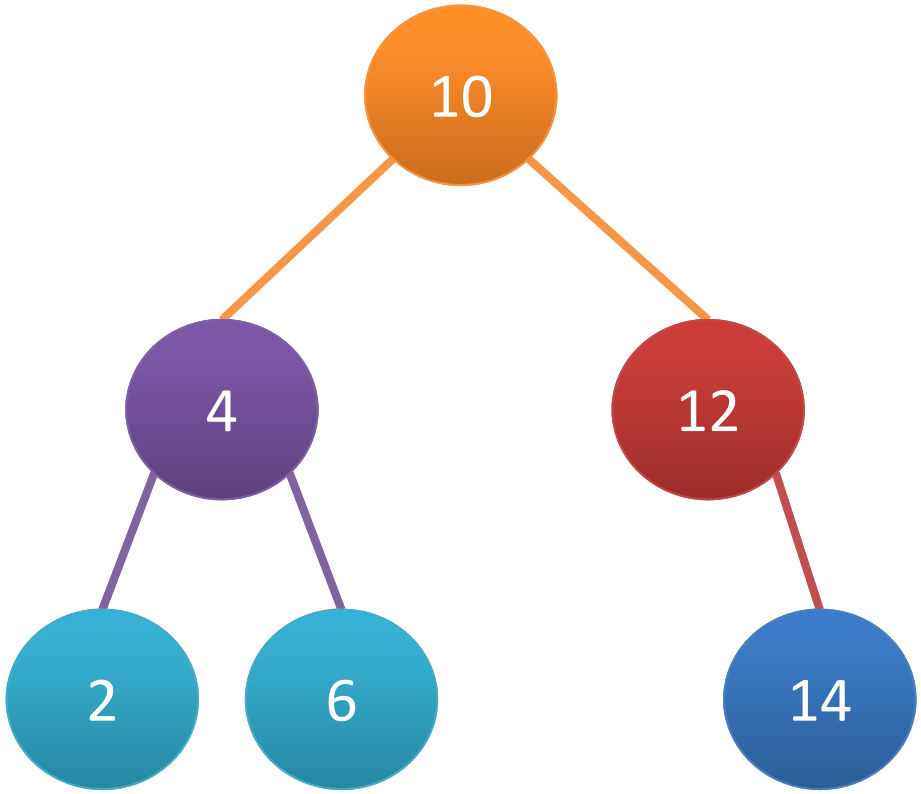
Trees in Real Life: Binary Search Trees

Let us try to find 11 in this binary search tree:



Trees in Real Life: Binary Search Trees

Let us try to find 11 in this binary search tree:



It isn't present in the BST. ☹

Trees in Real Life: Binary Search Trees

Two important things when searching for an item in a binary search tree (BST):

1. As we go down the tree, from one level to another, from parent to child, we discard *approximately half* the data each time.
2. In the *worst case*, we go down all the way to the leaves.

Trees in Real Life: Binary Search Trees

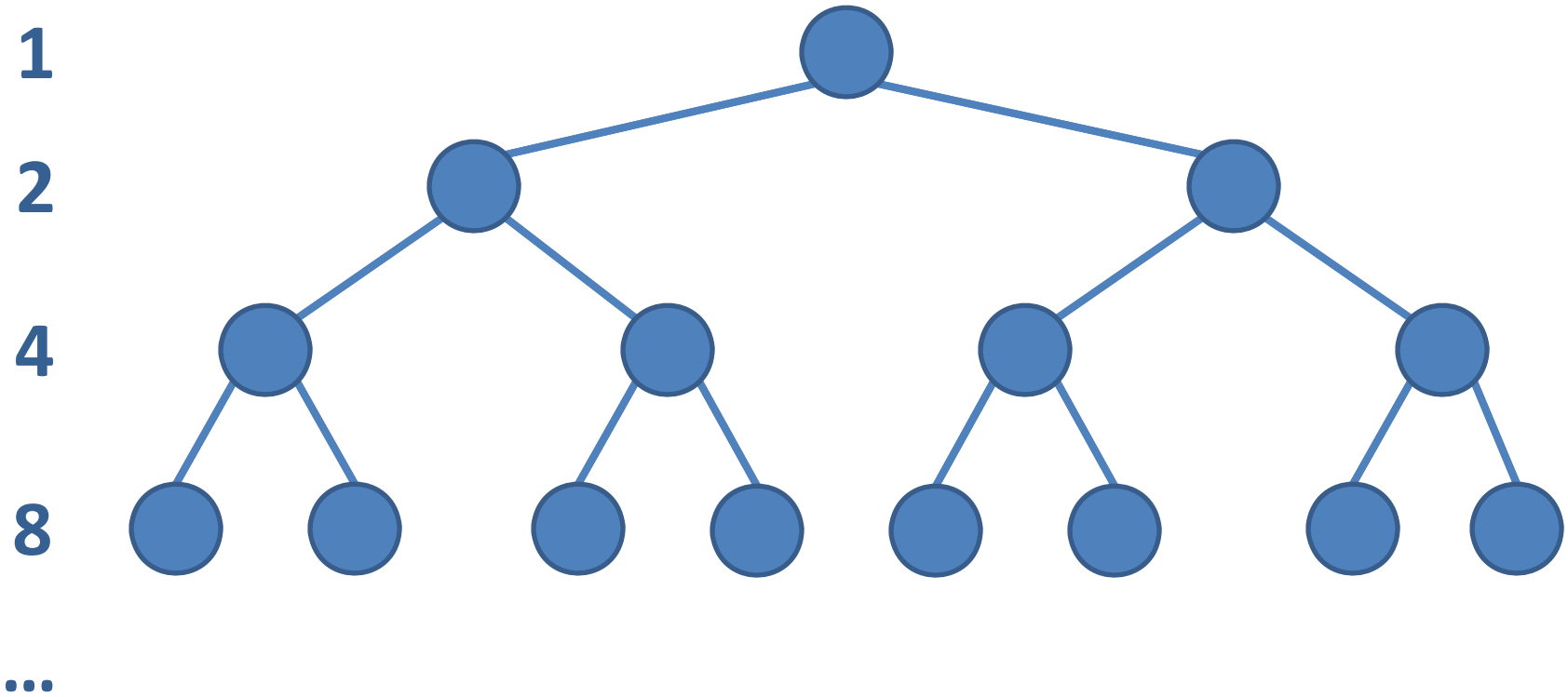
The ***height*** of a tree is the length of the *longest path* from the root to any leaf.

If there are n nodes in a binary search tree, and if the binary search tree is *balanced*, then the height of the tree is approximately $\log_2 n$.

Approximately as many nodes on the left of any node as there are on the right of that node.

LOGARITHMIC ALGORITHM

Why approximately $\log_2 n$?



LOGARITHMIC ALGORITHM

Why approximately $\log_2 n$?

For a tree of height h (or $h + 1$ levels), the total number of nodes is

$$1 + 2 + 4 + 8 + \ldots + 2^h = n$$

or, $2^{h+1} - 1 = n$

or, $h \approx \log_2(n)$

LOGARITHMIC ALGORITHM

Why approximately $\log_2 n$?

Alternatively, notice that when we go *down one level*, we are discarding ***approximately half*** the data.

Trees in Real Life: Binary Search Trees

In the *worst case*, we go down to the leaves, visiting each level of nodes ***once***.

There are approximately $\log_2 n$ levels.

The runtime to find an item in a binary search tree (BST) with n nodes is $O(\log n)$.

Exponentially faster than $O(n)$!

BST ADT

```
empty_bst = None
def make_bst(datum, left=empty_bst,
                    right=empty_bst):
    return (left, datum, right)

def bst_datum(b):
    return b[1]
def bst_left(b):
    return b[0]
def bst_right(b):
    return b[2]
```

CONSTRUCTOR

SELECTORS

BST ADT

`bst_is_leaf(b)` returns `True` if the BST is a leaf:

```
def bst_is_leaf(b):
    return bst_left(b) == empty_bst and
           bst_right(b) == empty_bst
```

BST ADT

```
def has_num(b, num):
    if b == empty_bst:
        return False
    elif bst_datum(b) == item:
        return ____
    elif bst_datum(b) > item:
        return has_num(____________,
                       num)
    return has_num(____________,
                   num)
```

BST ADT

```
def has_num(b, num):
    if b == empty_bst:
        return False
    elif bst_datum(b) == item:
        return True
    elif bst_datum(b) > item:
        return has_num(bst_left(b),
                       num)
    return has_num(bst_right(b),
                   num)
```

CONCLUSION

- ITrees are useful for representing general tree-structures.
- Binary search trees allow us to search through data faster.
- ***Preview:*** Object-oriented programming: a brand new paradigm.

GOOD LUCK TONIGHT!

http://www.whosawesome.com/images/awesome.jpg

EXTRA: TUPLES VERSUS IRLISTS

Tuples and IRLists are two *different* ways of representing *lists of items*: they are two different data structures, each with their own advantages and disadvantages.

You will compare them further in CS61B, but here is a preview.

EXTRA: TUPLES VERSUS IRLISTS

IRLists are recursively defined, tuples are not.

This gives IRLists an advantage when you want to perform a naturally recursive task, such as mapping a function on a sequence.

EXTRA: TUPLES VERSUS IRLISTS

Which of the following methods is better?

```
def map_irlist(fn, irl):
    return make_irlist(fn(irlist_first(irl)),
                       map_irlist(fn,
                                  irlist_rest(irl)))

def map_tuple(fn, tup):
    results = ()
    for item in tup:
        results = results + (fn(item),)
    return results
```

EXTRA: TUPLES VERSUS IRLISTS

`map_irlist` is better than `map_tuple`.

`map_tuple` creates and discards temporary tuples, since *tuples are immutable*.

`map_irlist` "tacks on" new items to the front of an existing (and growing) IRList.

EXTRA: TUPLES VERSUS IRLISTS

Tuples are unchanging sequences of data, which gives them an advantage when all you want to do is grab information from a sequence.

EXTRA: TUPLES VERSUS IRLISTS

Which of the following methods is better?

```
def getitem_tuple(tup, pos):
    return tup[pos]

def getitem_irlist(irl, pos):
    if pos == 0:
        return irlist_first(irl)
    return getitem_irlist(irlist_rest(irl), pos-1)
```

EXTRA: TUPLES VERSUS IRLISTS

`getitem_tuple` is better than `getitem_irlist`.

Indexing into a tuple takes (*on average*) constant time, but indexing into an IRList takes linear time.